Progress in Profiling and Parallelizing the ACES II Program System

Anthony D. Yau, Piotr Rozyczko,
S. Ajith Perera, Rodney J. Bartlett,
Quantum Theory Project
University of Florida
Gainesville, FL 32611

K. Szalewicz
Department of Physics and Astronomy
121 Sharp Laboratory
University of Delaware
Newark, DE 19716

N. Nystrom
Pittsburgh Supercomputing Center
Mellon Institute Building
4400 Fifth Avenue
Pittsburgh, PA 15213

J. P. Blaudeau
ASC/HP, 2435 5th Street, Bldg. 676
Wright-Patterson Air Force Base, OH 45433-7802

M. Zottola
1201 Technology Drive
Suite 206
Aberdeen, MD 21001

Abstract

The program that calculates the coupled-cluster electron correlation energy in the ACES II Program System was profiled and analyzed. With this information, we began parallelizing the most expensive subroutines with message passing and multithreading via MPI and OpenMP, respectively. We discuss the nature of the most important bottlenecks and our parallelization strategy for each. Initial timings for these parallel subroutines are also presented.

Introduction

Under 25 years of support from the DoD basic research agencies, AFOSR, ONR, and ARO; Rod Bartlett's research group at the University of Florida Quantum Theory Project formulated and implemented into general-purpose programs coupled-cluster (CC) methods for highly accurate electronic structure calculations. The ACES II program was written in the early 90's [1,2,3]. The goal was to create a tool for the routine application of CC methods for open- and closed-shell molecules; for ground, excited, and ionized states; for molecular structure and spectra (IR, Raman, UV-vis, NMR, ESR, PES); for transition state and activation barrier determinations; and for molecular properties; while using real abelian symmetry [4,5]. ACES II is still at the forefront in terms of functionality and efficiency.

The standard CC algorithm solves a set of coupled amplitude equations. In general, a scheme based on molecular orbitals (MO) is faster than one based on atomic orbitals (AO) since the MO target matrix (T amplitudes) is at least 75% smaller than the AO target matrix. The disadvantage of using an MO-based scheme is the volume of transformed integrals needed for evaluating the T amplitudes.

One way to decrease the sizes of the storage files is to transform only the smallest MO integrals while leaving the larger quantities for on-the-fly transformation with stored or direct AO integrals. This is often referred to as "using the AO basis" while it is implied that the T amplitudes are defined in the MO basis. The disadvantage of AO basis schemes is the time spent evaluating all the contributions a single integral makes to the amplitudes. For large systems of chemical interest, we need to use the AO basis for the virtual-virtual-virtual-virtual MO integrals.

Due to recent demands for calculations on larger molecules (20-30 atoms) and to the need for faster throughput in calculations, we have begun parallelizing the CC algorithm in the ACES II program. The initial phase of this effort, contracted by PET under the CCM CTA, attempts to micro-parallelize the existing CC single-point energy program wherever possible rather than to build a new software package from scratch. Before programming could begin, a thorough profiling of the existing CC implementation was required in order to identify the bottlenecks and set directions for parallelization.

The software described here was developed on a cluster of IBM RS/6000 workstations each with four 375 MHz POWER3 CPUs, 3 GB of main memory, and 18 GB of local disk space. The nodal configurations are similar to the IBM SP clusters installed at the NAVO, ARL, and ASC MSRCs with two exceptions. Our development cluster is connected with fast ethernet and the MSRCs use high-performance switches. Although porting the code will only decrease communication costs, a much more problematic difference is the lack of large local storage. The current parallelization strategy intentionally avoids distributed or remote storage strategies, and forcing the current prototype to access remote files will severely degrade performance.

Profiling

Initial attempts at timing the code revolved around the common profiling utility gprof. Most commercially available compilers provide a mechanism for creating profiler data files after running a binary executable. The data files may then be analyzed by utilities, which generate statistics on CPU usage by each subroutine during the program's execution. We discovered these analyses to be largely inadequate since the gprof call graph assignments are based on averaged times from a histogram. For example, subroutine A may call DGEMM, a double precision matrix-matrix multiply [6], 99 times for a total cost of 1 second while subroutine B calls DGEMM once for a cost of 9 seconds. We are more interested in parallelizing the 9-second DGEMM than the 99 0.01-second DGEMMs. The gprof dependency graphs cannot tell the difference. The averaged results would indicate that 99% of the CPU time is spent in the DGEMMs called by subroutine A.

A profiler can justify this behavior with the fact that the program has no control over other user or system processes, which may be competing for shared system resources. If a compute-intensive process runs simultaneously with the program being profiled, then a wallclock profiler will unfairly weight whatever routines are called during this interval. This led us to create our own wallclock or realtime profiler, and we have been careful to time jobs run only with dedicated system resources.

For the systems of current interest – C_1 symmetry with many basis functions using the AO basis scheme – 20% to 30% of the total CCSD wallclock time is spent generating intermediate quantities used to determine the T amplitudes. Another 40% to 60% is spent generating the amplitudes. Further analysis of these bulk operations shows that less than 10 routines consume close to 90% of the total CCSD wallclock time.

Table 1 lists the main subroutines in the ACES coupled-cluster member executable, called vcc, and the average percentage of wallclock time each one consumes over five molecular systems. From the table, the routines that individually account for more than 2% of the wallclock time are RDAO, CNTRCT, T1W1AB, RNGPRD, T1RING, and DODIIS. Other routines that we may consider for micro-parallelization are QUAD1, T1INT2B, and LADAB.

GEMM-bound

CNTRCT, T1W1AB, and RNGPRD each wrap a single yet very large DGEMM. It is common for each call to take tens or hundreds of seconds. T1RING is a wrapper for three routines, each of which calls DGEMM tens or hundreds of thousands of times, but each call lasts only a few milliseconds.

I/O-bound

The traditional answer for why vcc might be expected to suffer from I/O involves the processing of massive data files needed by each CC iteration. However, comparing total wallclock time to total user time for a dedicated node shows that the performance of vcc is more affected by faster CPUs and memory accesses than faster I/O subsystems.

Processor-bound

Processor-bound algorithms are ones in which the speeds and sizes of the memory and memory caches are the primary bottleneck [7]. Accessing arrays repeatedly in non-unit stride is a prime example of a processor-bound operation. With a reimplementation of an algorithm, it is possible to remove the architecture dependence and instead make the operation compute-bound. Compute-bound algorithms are the best ones to have (in the current technology market) since CPU speeds increase at a much more accelerated rate than the speeds of memory and I/O buses.

RDAOIJKL is responsible for loading and contracting all the AO integrals with the T_2 amplitudes from the previous CC iteration. The product array is named Z_2, and this routine alone consumes 40% to 60% of the CCSD wallclock time. RDAOIJKL is processor-bound. It is true that this routine reads in all the AO integral files, but these read operations are sequential and unformatted. The aspect of the algorithm that slows it down so much is that it almost randomly updates columns in a very large matrix. Less than 1% of the wallclock time is spent reading in the integrals. 1% to 3% of the time is spent looking up the read and write indices of the T_2 and Z_2 columns for each integral contraction. The remaining time is spent performing the $Z_2[]\mathrel{+}=X*T_2[]$ contractions.

Parallelization

GEMM-bound

There are hundreds of distinct DGEMM calls in the vcc code. However, less than 10 are responsible for more than 90% of the wallclock time attributed to matrix multiplies. Attempts at linking to a fully parallel or multi-threaded BLAS library [8] have been disappointing. So much time is added by the previously insignificant DGEMMs that the few that might benefit from MP or MT are not able to compensate properly for the extra overhead.

The easiest way to decompose a matrix-matrix multiply is by dividing the columns (rows of the transpose) of the second/right matrix over the number of threads, nodes, or both. This will introduce cache coherence problems for MT DGEMMs, but the effect is much less than dividing the rows of the first/left matrix over the number of threads. The performance of a DGEMM call divided over two threads with column blocking is shown in Table 2. In any case, there are countless studies of parallelizing matrix-matrix multiplies. We do not intend to reinvent this wheel, so we will use an off-the-shelf parallel DGEMM if column blocking proves to be too inefficient.

I/O-bound

For our initial test systems on dedicated nodes, the percentage of CPU time to wallclock time never dropped below 70% for CCSD. Analyzing the CPU usage during a vcc execution shows that there are times when the processor usage drops below 2% for sustained periods. The routine that is responsible for this has yet to be found; however, every node currently requires a full file set. Without an interprocedural analysis (IPA) [9] or a serial reimplementation, both of which will be performed before the project ends, this behavior will not change.

Processor-bound

Since beginning this project, RDAOIJKL has been reimplemented twice. The first optimization removed a very costly loop of conditionals over the contraction matrix. The second optimization treats integrals with four common symmetries (denoted IIII) separately from those with differing symmetries (such as IJIJ, IIJJ, and IJKL).

Distributing each integral batch over the number of nodes allows the RDAO routines to achieve near-linear scaling assuming the processor subsystems are the same in all nodes. The theoretical limit is the number of integrals (600 by default) in a batch. We have experimented with further dividing the write-domain (columns in the Z_2 matrix) over available threads. The wallclock time decreases with the thread count, but the scaling is not linear since there is no guarantee of perfect (or even decent) load balancing while processing an integral batch. To further reduce the effect of cache conflicts, we intend to implement a leading dimension to the Z_2 matrix that is a multiple of the size of a cache line. The performance of RDAOIIII divided over 2 MPI processes and 2 OpenMP threads is shown in Table 2.

The disadvantage to this distribution over nodes is the need to allreduce (global reduction followed by global broadcast) the large Z_2 matrix after processing the integrals. This communication is quite costly; however, the bottleneck is the network bandwidth and not the communication latency.

Future Work

The two main limiters from achieving 100% parallel efficiency or linear scaling are serial bottlenecks and interprocess communication (including synchronization). With a parallel CC algorithm, there will always be some amount of redundant serialization (e.g., determining low-cost one-particle intermediates) and some amount of synchronized communication (e.g., broadcasting the new amplitudes between the nodes before the next CC iteration). However, there are multiple ways to parallelize most algorithms and we must rely on real-world profiling results to guide our efforts.

Decrease Communication

From an interprocedural analysis of various routines, it may be possible to communicate reduced quantities at a lower cost than to communicate shared quantities the moment they become available (a consequence of micro-parallelization). An example of such an analysis shows that the temporary T_2 contributions in the partially back-transformed state are fully transformed before being added into the new T_2 amplitudes. The full MO representation of an (AB,IJ) type operator will always contain fewer elements than a full AO representation. Instead of calling MPI_Allreduce on the full Z_2 matrix after the rdao routines, the code could wait until the retransformation and allreduce fewer T_2 contributions at a lower cost based on the communications bandwidth.

Tune Serial Routines

One of the disadvantages of using research-oriented software is the initial use of naive implementations with the intent to optimize later. If the programmer is drafted into other projects, that optimization may never take place.

We have discovered quite a few routines that could benefit from serial reimplementation. DODIIS is one routine that consistently consumes 1% to 3% of the total wallclock time. The bottleneck in that routine involves forming a matrix of dot products whose factors track the convergence of the T amplitudes over multiple iterations. The current algorithm is limited by available memory and I/O bandwidth. An alternate serial algorithm could be implemented with new code, which will be definitely faster, or a parallel algorithm could be implemented with a new dependence on communication latency.

Distribute Computation from IPA

The determination of T amplitudes at one iteration only depends on the amplitudes from the previous iteration and constant one- and two-particle operators and integrals. With a massive IPA, it is technically possible to distribute the amplitude determination over the compute nodes such that only the intermediate quantities needed for each subset of amplitudes are computed locally. In principle, this approach should limit the communication to a single allgather at the end of each CC iteration; however, the nature of the electron correlation problem suggests otherwise if we choose not to recompute common intermediates among the nodes. Such an algorithm will have to be studied and tested for performance.

References

1 ACES II is a program product of the Quantum Theory Project, University of Florida. Authors: J.F. Stanton, J. Gauss, J.D. Watts, M. Nooijen, N. Oliphant, S.A. Perera, P.G. Szalay, W.J. Lauderdale, S.A. Kucharski, S.R. Gwaltney, S. Beck, A. Balkov, D.E. Bernholdt, K.K. Baeck, P. Rozyczko, H. Sekino, C. Hober, and R.J. Bartlett. Integral packages included are VMOL (J. Almlof and P.R. Taylor); VPROPS (P. Taylor); ABACUS (T. Helgaker, H.J. Aa. Jensen, P. Jorgensen, J. Olsen, and P.R. Taylor).

2 J. F. Stanton, J. Gauss, J. D. Watts, W. J. Lauderdale, and R. J. Bartlett, "The ACES II Program System," IJQC: Quant. Chem. Symp. **26**, 879-894 (1992).

3 R. J. Bartlett and J. D. Watts, “ACES II,” in Encyclopedia of Computational Chemistry (John Wiley & Sons, 1999).

4 R. J. Bartlett, "Coupled-Cluster Approach to Molecular Structure and Spectra: A Step Toward Predictive Quantum Chemistry," J. Phys. Chem. **93**, 1697-1708 (1989).

5 R. J. Bartlett, in Modern Electronic Structure Theory, Part II, edited by D. R. Yarkony (World Scientific, Singapore, 1995), p. 1047.

6 J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, "A set of Level 3 Basic Linear Algebra Subprograms," ACM Trans. Math. Soft. **16**, 1-17 (1990)

7 G. F. Pfister, In Search of Clusters, 2nd ed. (Prentice Hall PTR, Upper Saddle River, NJ, 1998), p. 154.

8 C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage," ACM Trans. Math. Soft. **5**, 308-323 (1979).

9 F. E. Allen, in Proceedings IFIP Congress, 1974 (North-Holland, 1974), pp. 398-402.

Table 1. Average wallclock percentages for vcc subroutines from 5 RHF, C_1, AO basis systems.

ROUTINE NAME	PERCENT OF TOTAL WC TIME		ROUTINE NAME	PERCENT OF TOTAL WC TIME
ZERSYM	0.2%		FEACONT	0.5%
T1RING	3.7%		FMICONT	0.2%
CNTRCT	10.1%		FMECONT	0.4%
SUMSYM	0.1%		T1INT2A	0.3%
F2TAU	0.0%		T1INT2B	1.6%
GETLST	0.8%		T1INT1	0.4%
QUAD1	1.1%		LADAA	0.0%
QUAD2	0.4%		LADAB	1.2%
QUAD3	0.3%		T2TOAO	0.6%
MAKFME	0.2%		RDAO	52.9%
T1W1AA	0.0%		Z2TOMO	0.9%
T1W1AB	5.4%		ZIAAO	0.1%
T1INW2	0.0%		T12INT2	0.7%
			RNGPRD	11.1%
			SUMRNG	0.3%
			ERNGAA	0.0%
			ERNGAB	0.2%
			SUMSYM	0.2%
			E4S	0.2%
			NEWT2	0.4%
			PHASE3	0.5%
			DODIIS	3.3%

Table 2. Preliminary results comparing average wallclock times in parallel (2 nodes with 2 CPUs each) with average wallclock times in serial.

ROUTINE NAME	1/1 TIME (s)	2/2 TIME (s)	ACTUAL SPEED-UP
RDAOIIII	955.8	277.5	3.4
MT_DGEMM	567.9	294.7	1.9